LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-JRNL-503653

Grouper: A Compact, Streamable Triangle Mesh Data Structure

M. Luffel, T. Gurung, P. Lindstrom, J. Rossignac

October 7, 2011

IEEE Transactions on Visualization and Computer Graphics

Disclaimer

Grouper: A Compact, Streamable Triangle Mesh Data Structure

Mark Luffel, Topraj Gurung, Peter Lindstrom, and Jarek Rossignac

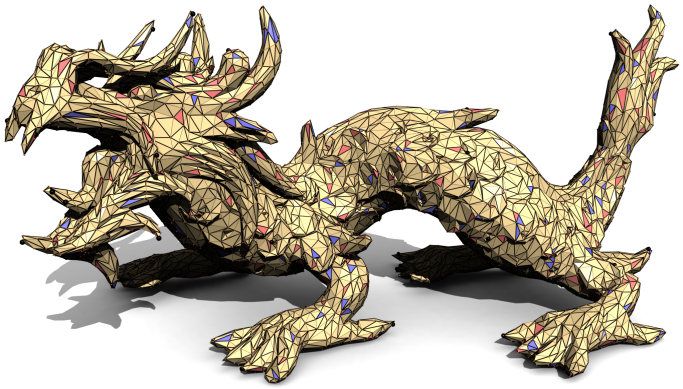


Fig. 1. Grouper represents a triangle mesh as groups of vertices and triangles stored as fixed-size records, most of which encode two adjacent triangles and one incident vertex. A VTT group (tan: 93%) represents one vertex and two adjacent triangles incident upon it; a VT group (blue: 3%) represents one vertex and one incident triangle; a T group (red: 4%) represents one triangle; and a V group (black: 1%) represents one vertex. Thick edges separate groups; thin edges separate triangles within the same group.

Abstract—We present Grouper: an all-in-one compact file format, random-access data structure, and streamable representation for large triangle meshes. Similarly to the recently published SQuad representation, Grouper represents the geometry and connectivity of a mesh by grouping vertices and triangles into fixed-size records, most of which store two adjacent triangles and a shared vertex. Unlike SQuad, however, Grouper interleaves geometry with connectivity and uses a new connectivity representation to ensure that vertices and triangles can be stored in a coherent order that enables memory-efficient sequential stream processing. We present a linear-time construction algorithm that allows streaming out Grouper meshes using a small memory footprint while preserving the initial ordering of vertices. As part of this construction, we show how the problem of assigning vertices and triangles to groups reduces to a well-known NP-hard optimization problem, and present a simple yet effective heuristic solution that performs well in practice. Our array-based Grouper representation also doubles as a triangle mesh data structure that allows direct access to vertices and triangles. Storing only about two integer references per triangle—i.e. less than the three vertex references stored with each triangle in a conventional indexed mesh format—Grouper answers both incidence and adjacency queries in amortized constant time. Our compact representation enables data-parallel processing on multicore computers, instant partitioning and fast transmission for distributed processing, as well as efficient out-of-core access. We demonstrate the versatility and performance benefits of Grouper using a suite of example meshes and processing kernels.

Index Terms—Mesh compression, mesh data structures, random access, out-of-core algorithms, large meshes.

1 Introduction

Triangle meshes are the most common representation of surfaces in computer graphics and computational science. A typical representation of a triangle mesh is as a table of vertices and a table of indices to the vertices, where each consecutive triplet of indices represents one triangle. This "indexed mesh" representation supports tasks such as rendering, but operations that require knowing which triangles are adjacent to one another necessitate a linear search over all triangles. To provide constant-time access to adjacent elements it is common to store additional adjacency references. Such information is required by tasks such as tracing the intersection between surfaces, solving differential equations defined on a surface, and by most other geometry processing and analysis applications.

For massive meshes that do not fit in main memory, the construction of adjacency relationships from an indexed mesh is a difficult task. A typical approach requires a series of external sorts [1], temporary storage several times that of the final data structure (sometimes tens of gigabytes), and several hours to construct [2–5]. The final data structure is usually a factor of 2–3 larger than an indexed mesh. Working with such a data structure often involves on-demand paging and explicit caching of portions of the mesh.

As an alternative to such out-of-core data structures, simple data analysis and geometry processing tasks can often be implemented as stream kernels that make a single sequential pass over the mesh. A *streaming mesh* [6] interleaves vertices and triangles and encodes when mesh elements are last referenced, allowing proactive deallocation and in-memory random access to a small active set. Such mesh formats, which store only little more information than an indexed mesh, can easily be constructed on the fly as part of a mesh generation process without using any intermediate disk. However, for tasks that require adjacency information, an in-memory partial data structure must be built and maintained by the stream kernel, which must map

- *Mark Luffel, Topraj Gurung, and Jarek Rossignac are with the Graphics, Visualization, and Usability Center (GVU), Georgia Institute of Technology. {mluffel,topraj,jarek}@cc.gatech.edu.*
- *Peter Lindstrom is with the Center for Applied Scientific Computing (CASC), Lawrence Livermore National Laboratory. pl@llnl.gov.*

Manuscript received 31 March 2011; accepted 1 August 2011; posted online 23 October 2011; mailed on 14 October 2011.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org.

global vertex indices to the in-core data structure. This data structure is then discarded, and the work required to build it is replicated each time the mesh is processed, for instance by each module in a pipelined computation. Moreover, because stream processing relies on sequential I/O, it is not well suited for tasks that require only select subsets of the mesh, such as localized queries and data-dependent traversals, as finding those subsets may require visiting the whole mesh repeatedly.

To provide the flexibility of random access while achieving the resource efficiency of stream processing, we propose a new all-in-one (1) binary format, (2) adjacency data structure, and (3) streamable representation, called *Grouper*. Our format, which builds upon the *SQuad* data structure [7], partitions the mesh into groups represented by fixed-length records that store up to one vertex and two incident triangles. The mesh connectivity is represented as pointers between adjacent groups. These pointers form loops in the dual graph. This information is sufficient to determine both adjacency and incidence, allowing meshes to be represented using only about two indices per triangle (i.e. less than the three needed in an indexed mesh). Grouper stores geometry and connectivity interleaved, and uses a more general representation than SQuad that allows meshes to be both streamed in and out of memory.

We show how our Grouper data structure can be constructed efficiently from an incidence-based streaming mesh format using only a small memory buffer and localized reordering of triangles (to decrease storage). The result may be streamed directly to another process or stored on disk. In the former case, we show how *windowed streaming* of a Grouper mesh, which uses a fixed-size in-memory buffer, eliminates the need to construct connectivity on the fly, as normally required by stream kernels that process incidence-based streaming formats. Because Grouper uses fixed-size records, it supports direct access to incidence and adjacency for navigating the mesh. Using memory mapped I/O, out-of-core support may be provided transparently to applications. Moreover, we show how data-parallel processing can be achieved using OpenMP [8] for those tasks that make sequential passes over the mesh, allowing substantial out-of-core processing speedups over prior methods.

2 Prior Art

Grouper brings together two threads of research: streaming meshes for out-of-core computations and compact data structures for random-access mesh processing (exemplified by SQuad). We review these two types of representations as well as external memory data structures in the following sections.

2.1 Compact Connectivity Representations

SQuad [7] is a triangle mesh data structure that provides constant-time adjacency queries and makes efficient use of memory, storing connectivity data in slightly more than 2 references per triangle (rpt). SQuad organizes most triangles in pairs that are matched with a single incident vertex. It stores adjacency relationships between triangles, but does not store vertex indices. Instead it infers incident vertices by examining a small set of candidates that are found by following loops in the dual graph. Our Grouper data structure builds upon these ideas. Although SQuad preserves vertex ordering, any unmatched triangles must be stored separately (e.g. at the end of the connectivity array), which degrades coherence and makes SQuad meshes difficult to stream both in and out of memory.

The LR data structure [9] is twice as compact as SQuad, storing about 1 rpt for connectivity, but is not suitable as a streaming representation because its construction imposes a global ordering of the vertices of a mesh along a nearly Hamiltonian cycle. This ordering may be incompatible with the one in which the mesh is generated, in which case multiple streaming passes or an external sort is needed to reorder the mesh. The Zipper improvement of LR stores only about 6 bits per triangle [10], but suffers from the same limitations. To be of practical use, we seek a data structure that can be built on the fly, concurrently with the mesh generation or processing.

The Tripod data structure [11] also represents triangle mesh connectivity. Its construction algorithm operates by contracting edges starting from a seed triangle, resulting in labels at each corner, and identifying three canonical outgoing edges per vertex, from which the other incident edges and faces can be inferred. This linear-time algorithm permits a single-pass streaming implementation, but the resulting data structure, at 3 rpt, is less compact than SQuad and can only represent genus zero meshes. The construction of Tripod requires additional storage for an auxiliary data structure that supports adjacency queries and a data-dependent traversal.

2.2 External Memory Data Structures

A number of external memory data structures for triangle meshes have been proposed, primarily for interactive visualization. To handle very large meshes, these data structures often support multiresolution adaptive refinement. The predominant approach is to partition the mesh into chunks of many triangles that are paged in from disk in atomic units [2–5, 12]. Memory management and I/O are usually handled explicitly by the application. Recognizing the performance bottleneck associated with I/O, recent work has focused on compressed representations [13–15] that support local decompression for access to vertices and triangles. Because these techniques use variable-length coding, the mesh is again partitioned into chunks to amortize the cost of specifying the locations of vertices and triangles in the file across a larger portion of the mesh. The decompressed mesh is usually cached in a conventional triangle mesh data structure, along with adjacency information derived from the compressed format. Although not as compact as these compressed formats, our Grouper representation uses fixed-length records and therefore supports random access at a much finer granularity. Furthermore, accesses involve only array lookups, and require no decoding or explicit caching.

2.3 Streaming Meshes

Streaming out-of-core computations process a mesh via sequential I/O, maintaining only a small portion of the mesh in core, on which random access is possible. Because resolving references from triangles to vertices in an indexed mesh typically requires that all vertices be stored and can be looked up in memory, early work on streaming assumed that the mesh was represented as *triangle soup*. Each triangle in a triangle soup is represented as a triplet of vertex positions (i.e. with no indices), allowing the triangles to be processed independently one at a time. Chiang and Silva [16] showed how a large indexed mesh can be converted to triangle soup using a series of external sorts, thereby stripping the mesh of its connectivity information. For stream processing tasks that require connectivity, this information has to be recovered on the fly, e.g. using geometric hashing on vertex coordinates [17–19]. However, if the surface has boundaries, the full connectivity around a vertex cannot be known until the entire stream has been processed. To avoid the problems of working with triangle soup, Isenburg et al. [20] suggested using a compressed format from which both local connectivity and geometry can be extracted. However, construction of this format requires an expensive out-of-core preprocess.

Streaming Meshes [6] are an out-of-core representation of indexed meshes that provide connectivity and geometry without the need to first buffer all vertices. A streaming mesh may be thought of as a decorated indexed mesh that interleaves vertices and triangles—to provide a local view of a portion of the mesh—and that supports garbage collection via reference counting—to limit the memory footprint. Instead of storing reference counts, a streaming format usually associates *finalization tags* with the vertices that certify that no future triangle in

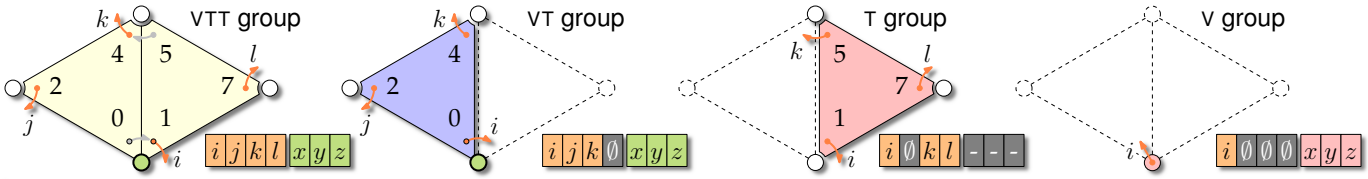


Fig. 2. Grouper represents vertices and triangles in one of four configurations. The green shaded vertices are matched with these triangles, and the orange arrows represent swing pointers $c.z$ to corners of other groups or to one The V group has no triangle, so we associate its vertex with a "virtual" corner. Within a group, corners are assigned in-group indices from zero to seven, with 0 identifying the corner matched with the vertex (or the "virtual" corner in V groups, described in Section 3.1).

the stream references a particular vertex, e.g. using one extra bit per vertex reference. This allows stream kernels to deallocate vertices when no longer needed, making it possible to process meshes with billions of triangles on off-the-shelf desktop computers. Streaming meshes may be compressed and decompressed on the fly [21], making the transfer of large meshes more feasible, and if secondary storage is very slow, improving the performance of mesh processing. A large number of geometry processing tasks have been adapted to streaming based on the streaming mesh representation, including Delaunay triangulation [22], remeshing [23], topological feature extraction [24], mesh simplification [25], surface reconstruction [26, 27], sampling [28], and mesh quality improvement [29], among others.

3 Grouper

Our compact Grouper representation supports on-the-fly streaming construction and processing, while also enabling constant-time random access to vertices, triangles, and their neighbors.

We present the details of the Grouper data structure in terms of triangle "corners," each of which associates a vertex with an incident triangle. We say that a vertex v and triangle t are **incident** if v is a vertex of t; two vertices are **adjacent** if they belong to the same triangle; and two triangles are adjacent if they share an edge. We use corners as iterators over the mesh. These can be manipulated by a set of traversal operators. We write operators in a post-fix dot notation, similar to object-oriented programming. The dot notation $c.t$ does not imply that c is a data record that contains a t field. For example, an operator may be implemented as a lookup into an array, or as a function to compute the value.

The operators applied to a corner c are illustrated in Fig. 3. They are $c.n$ (**next**) for circulating (clockwise) the corners within a triangle, $c.s$ (**swing**) for circulating (clockwise) the corners of a vertex, and $c.t$ (**triangle**) and $c.v$ (**vertex**) for extracting the triangle or vertex associated with c. We can combine these four operators to express other convenient operators. For example, **previous** can be written $c.p = c.n.n$ and **opposite** can be written $c.o = c.p.s.p$. Operators $v.c$ and $t.c$ return an unspecified incident corner of (respectively) a vertex or a triangle.

In an uncompressed data structure for general polygonal meshes, such as the half-edge representation [30], these operators are usually implemented using explicit pointers or indices. In our case, only $c.s$ is stored directly, while $c.n$ and $c.t$ are computed from c, and $c.v$ is inferred from $c.s$. This inference of $c.v$ is possible by matching each vertex with one of its incident corners, and by reordering the corners (and thus triangles) such that the vertex index may be inferred from the corner index.

Grouper partitions the triangles and vertices of a mesh into small groups of elements, of which there are four types: VTT, VT, T, and V (Fig. 1 and 2). The type indicates whether the group contains a vertex (V, VT, VTT), a triangle (T, VT), or two triangles (VTT). Using the construction algorithm described in this paper, most groups ($> 90\%$) are of type VTT, and consist of

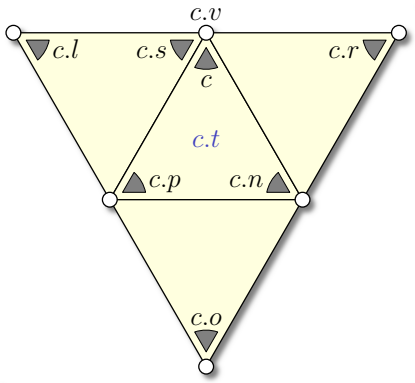


Fig. 3. The standard set of corner operators.

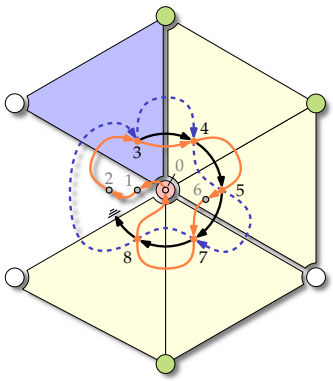
a pair of adjacent triangles that are matched with one of their shared vertices. In a mesh without boundary there are typically few V groups, and the remaining non-VTT groups are roughly equally divided into types T and VT.

Grouper stores a single array M of fixed-length records, each encoding adjacency and geometry data associated with a single group. Each record contains seven 32-bit values: four swing references i, j, k, l to adjacent groups, and three vertex coordinates x, y, z. We may optionally extend these records to support user-specified data (e.g. color, normals, material indices). Conceptually M is formed by interleaving a swing array S and a geometry array G. For simplicity of presentation, we will initially assume that S is a single contiguous one-dimensional array that stores i, j, k, l for each group. We will further assume that the mesh is manifold without boundary—an assumption we later relax.

The swing operators (called "swings") circulate clockwise around a vertex and form a closed loop. For each matched vertex v, we know that its swing loop must pass through the group containing v. Thus, for any corner c incident on v we can infer the result of the $c.v$ operator by traversing the loop with $c.s$. In particular, using a canonical labeling of corners within a group, only the "first" corner (corner 0) signifies a match. Because the average vertex valence in a mesh with low genus (relative to its number of triangles) is constant, the search through this swing sequence completes in expected (average) constant time.

3.1 Connectivity Operators

To define the mesh traversal operators for a Grouper mesh, we use a specific numbering of corners within a group (see Fig. 2). Each group covers a range of eight corner indices, of which at most six are used. Reserving eight rather than six corners allows us to implement some operators more efficiently (as bit shifts rather than divisions). Note that the unused corner indices do not cost us any storage space, because we do not allocate any per-corner data. Notice also that odd corner indices are



c	$c.z$	$c.h$	$c.s$
0	1	-	-
1	2	-	-
2	3	-	-
3	4	4	4
4	6	7	5
5	6	7	7
6	7	-	-
7	0	3	8
8	0	3	null

Fig. 4. Operators $c.s$ (black) and $c.h$ (dashed blue) are implemented in terms of operator $c.z$ (orange) stored in the swing table S. Actual corners are shown in orange; virtual corners in V groups and extensions are hollow. Orange arrows $c \rightarrow c.z$ traversing but not ending at a corner $c.s$ in a VTT group indicate that c and $c.s$ share $c.z = c.s.z$, since only one swing pointer per vertex is stored within a group.

assigned to the right triangle, which simplifies the index mapping from corners to triangles and vertices. Because in general not all groups are of type VTT, some vertex and triangle indices are unused. However, as we show in Section 7, this overhead usually amounts to less than 5%.

A corner with in-group index x (see Fig. 2) in group g has a global corner index $c = 8g + x$. Therefore, for a given integer corner index c, we may compute its group index as $c.g = c/8$ (where x/y denotes integer division) and its in-group corner index as $c \bmod 8$. For a manifold vertex v matched with a triangle, we could implement the $c.v$ operator by walking around v via $c.s$ until a matched corner (in-group corner 0) is reached. Clearly no such corner will be reached if v is not matched or if $c.s$ runs into a boundary (see Section 3.2). In addition, intra-group swings (i.e. from corners 0 and 5 in VTT groups) never lead to a matched corner, and therefore following such swings is wasteful when evaluating $c.v$.

To address these problems, we define two operators for internal use, $c.z$ and $c.h$, which are variants of $c.s$. The operator $c.z$ mimics $c.s$, but skips over intra-group swings and follows links to and from V groups and across boundaries. Hence, $c.z$ joins groups incident on $c.v$ in a circular linked list. We store in the swing table S such links $c.z$ rather than swings $c.s$. (Because of their similarity, we will sometimes refer to both $c.s$ and $c.z$ as swings around $c.v$.) We note that $c.z$ refers either to an actual *triangle corner* or to a *virtual corner* in a V group or extension (extensions are discussed in section 4.1.1), and we reserve one bit with each S table entry $c.z$ to indicate whether c is virtual. This *real* bit is true for each non-null entry of VTT, VT, and T groups and is false for all entries of V groups. For convenience, we also define another internal operator, $c.h$, that applies $c.z$ one or more times to skip over any virtual corners. These operators are illustrated in Fig. 4. We explain below how we implement $c.v$ and $c.s$ using $c.z$.

$c.s$ for intra-group swings involves only flipping the least significant bit of c: 0 swings to 1, and 5 swings to 4. Inter-group $c.s$ swings, on the other hand, require accessing the $c.z$ links, which are stored as four entries per group; one for each of the four vertices incident on the group (Fig. 2). To perform such a swing, we first map the in-group corner index $\{0, 1, 2, 4, 5, 7\}$ to the corresponding in-group swing index $\{0, 1, 2, 3\}$, which due to our corner numbering is easily computed as $c/2 \bmod 4$. Because these four swings are encoded in the one-dimensional array S as consecutive quadruplets, we may simply compute group links as $c.z = S[c/2]$. Below we will also refer to these links by group and in-group swing indices as $g.i = S[4g + 0]$, $g.j = S[4g + 1]$, and so on (see Fig. 2).

Because all groups incident on $c.v$ are joined by $c.z$ links and because intra-group swings never lead to a matched corner, by following $c.z$ around $c.v$ a traversal is guaranteed to reach the matching corner. Once we find the matching corner c, we compute $c.v = c/8$.

Our operators for meshes without boundaries are implemented as follows:

$$c.v = \begin{cases} c/8 & \text{if } c \bmod 8 = 0 \\ c.z.v & \text{otherwise} \end{cases}$$

$$c.s = \begin{cases} c \oplus 1 & \text{if } c \bmod 8 = 0 \wedge c.g.l \neq \emptyset \vee \\ & \quad c \bmod 8 = 5 \wedge c.g.j \neq \emptyset \\ c.h & \text{otherwise} \end{cases}$$

Here $x \oplus y$ denotes bitwise *exclusive or*, $x \wedge y$ denotes *logical and*, $x \vee y$ denotes *logical or*, and $\emptyset$ is a null value for distinguishing group types (see Fig. 2). The predicate in $c.s$ determines if the group $c.g$ is of type VTT, in which case swings from corners 0 and 5 map to corners 1 and 4, respectively, within the same group. Otherwise, we swing to a triangle corner in another group using $c.h$, skipping over any virtual corners.

Due to the grouping of vertices and triangles, the remaining operators are straightforward: $t.c = 4t + (t \bmod 2)$, and $c.t = 2(c/8) + (c \bmod 2)$. Notice that the assignment of even corners to the first triangle of a group and odd corners to the second triangle makes this mapping possible. For a matched vertex v, $v.c = 8v$. For unmatched vertices (whose group's k pointer is null) $8v$ gives the index of a virtual corner c, from which we follow $c.h$ to arrive at $v.c$. $c.n$ modifies only the lower three bits of c, and hence can be coded as a small lookup table.

When the connectivity array S and geometry array G are interleaved as a single mesh array M (as in Fig. 2), we must translate indices to the array S to corresponding M indices. (We do not, however, change what is stored in the S array.) We provide here a general translation assuming four S fields and n additional fields per group: $S[i] \mapsto M[i + n(i/4)]$, again assuming integer division. For instance, $S[i] \mapsto M[i]$ when $n = 0$, and $S[i] \mapsto M[2i - (i \bmod 4)]$ when $n = 4$.

When sequentially iterating over all vertices, triangles, or corners in the mesh by index, we must test if the index corresponds to a valid mesh element. (This is not necessary when starting a traversal from a valid element.) The following predicates test if a vertex v, triangle t, or corner c is valid:

$$\begin{aligned} valid(v): &\quad S[4v + 1].real \vee S[4v + 2] = \emptyset \\ valid(t): &\quad S[2t + 1].real \\ valid(c): &\quad c \bmod 8 \notin \{3, 6\} \wedge valid(c.t) \end{aligned}$$

where *real* denotes the bit indicating if a corner is real or virtual.

Finally, we implement $c.n$ and $c.p$ as the *exclusive or* between c and precomputed two-bit constants (these operators affect only the second and third least significant bits of the corner index c). This allows us to encode the transition table for $c.n$ as a packed 16-bit constant (and similarly for $c.p$) and to evaluate these operators in constant time using only bitwise shifts and logical operations with no memory accesses:

$$c.n = c \text{ XOR } \left(((2 \times \text{c639}_{16}) \gg (2 \times (c \text{ AND } 7))) \text{ AND } 6\right)$$

$$c.p = c \text{ XOR } \left(((2 \times \text{4b1e}_{16}) \gg (2 \times (c \text{ AND } 7))) \text{ AND } 6\right)$$

3.2 Boundaries

Whereas the application can freely visit all corners around an interior vertex from any other incident corner via $c.s$, boundary vertices require more care. Consider a boundary vertex v with incident corners $\{c_1, c_2, \ldots, c_m\}$, such that $c_i.s = c_{i+1}$. As a convenience to the application, we wish to match v with c_1

so that $v.c = c_1$ and the remaining incident corners on v may be visited via $c.s$. If we instead matched with any other corner c_i, then upon reaching c_m the application (which does not have access to $c.h$) would have to backtrack to visit the remaining corners $\{c_1, \ldots, c_{i-1}\}$ using the inefficient $c.u$ operator. Consequently, when v is a boundary vertex, we allow v to be matched only with c_1. If $c_1.t$ has already been matched, then we create a V group for v. Otherwise, we set $c_m.z = c_1$ to close the loop. Thus $c.s$ does not exist when $c.h.n.h.n \neq c$, in which case $c.s$ returns null instead of $c.h$. An implementation may for performance reasons choose to explicitly store whether $c.s$ exists, e.g. as a reserved bit in $c.z$. A similar approach was outlined in [7] to support both boundaries and non-manifold vertices.

As discussed above, we use a null value to indicate that the group does not have a swing in the corresponding position, and hence to distinguish the four group types (see Fig. 2). Toward this end, we make the observation that generally $c.z \neq c$, and therefore a relative swing of zero is not possible and can be reserved as a null value. One exception occurs for boundary vertices with one incident triangle, where in order to complete the swing loop we store a swing pointer from c to itself. In this case we store as part of the swing pointer a nonzero bit to mark that this is a real corner, and thus the stored value must in this case also be nonzero.

3.3 Relative Indexing

Our implementation of Grouper uses relative indexing for the swing pointers, i.e. we store in the swing table S with corner c the difference between $c.z$ and c rather than the absolute index $c.z$. In practice we store $c.z - 2(c/2)$ to ensure that we reach the same inter-group corner $c.z$ from in-group corners 0 and 1 (and similarly for corners 4 and 5).

Although a subtle difference, relative indexing has some desirable advantages for out-of-core and parallel processing. In particular, by interleaving connectivity and geometry and by using relative indexing, any contiguous subsequence $M' \subseteq M$ of the mesh array M is also a valid mesh, with the exception of those vertices $v \in M'$ whose swing loops extend outside M' and therefore cannot be dereferenced. This can trivially be remedied by slightly expanding M', making it possible to partition and process (slightly overlapping) pieces of M in parallel without having to remap indices. Conversely, using concatenation one may combine independent Grouper streams, e.g. pieces of an isosurface extracted in parallel from a partitioned domain. (As in other mesh representations, we would also have to identify and stitch vertices and edges shared between pieces if a water tight mesh is desired.) This feature combined with the compactness of Grouper makes it well suited as an interchange format for distributed processing.

3.4 Comparison with SQuad

For the reader's convenience, the rest of this paper is self-contained, and familiarity with SQuad [7] is not required. For the benefit of readers familiar with SQuad, we compare in this subsection SQuad with Grouper. At a high level, Grouper has (1) a simpler set of traversal operators, (2) the ability to represent a wider class of meshes, and (3) better memory locality that facilitates stream processing and enables streaming construction.

Operators Both data structures represent connectivity in terms of an array S of swing references between corners. In SQuad, these swing references are between "quad corners." That is, triangle corner pairs $\{0, 1\}$ and $\{4, 5\}$ in Fig. 2 are treated as a single quad corner. Because applications work with triangle corners, SQuad requires back and forth translation between triangle and quad corners, which complicates the implementation. Grouper, on the other hand, is based entirely on triangle corners. As in SQuad, our new data structure avoids visiting triangle corners 1 and 4 when searching for a matched corner, which conceptually is equivalent to converting to quad corners—but using only a simple rightward bit shift. Unlike in SQuad, the S table in Grouper stores triangle corners, and hence no quad-to-triangle corner conversion is needed.

Representable Meshes SQuad does not support the notion of unmatched vertices (V groups), but assumes that all vertices can be matched. For certain meshes (e.g. those containing isolated triangles with no neighbors, or triangle strips with more vertices than triangles) not all vertices can be matched (since a triangle may be matched with only one of its vertices). The introduction of a V group allows us to represent any manifold mesh with (or without) boundary.

Streamability SQuad stores two dense arrays: G containing vertex coordinates and S containing swing references. The n_V vertices and matched triangles are stored as the first n_V records of the S array, and are followed by the remaining unmatched triangles (T groups). Storing these triangles at the end of the array degrades locality and results in high-span layouts [6] that can be difficult to stream. Moreover, the predicate $c \bmod 8 = 0$ for identifying matched corners must be supplemented in SQuad to test if c lies in an unmatched triangle. In Grouper we interleave the G and S arrays and make use of a special T group that does not use corner 0, allowing us to store these triangles near their incident vertices and adjacent triangles. In particular, this allows us to stream out groups using a small memory footprint, whereas the original SQuad construction requires the whole mesh, including adjacency information, to reside in memory. We discuss our construction algorithm next.

4 STREAMING I/O

To handle large meshes, it is necessary that we construct and process Grouper streams without keeping the entire mesh in memory. Here we present a construction algorithm that matches triangles with vertices and outputs records of the Grouper representation while keeping only a small piece of the input mesh in memory. Our streaming writer reconstructs adjacency information for triangles on the fly, and so can accept as input an indexed streaming mesh [6], i.e. an interleaved sequence of vertices, triangles, and finalization tags. Because the algorithm operates on streaming meshes, it can be spliced into a processing pipeline, running concurrently with its source and sink processes, without saving to an intermediate file. Although we preserve the ordering of vertices specified by the application, it is often necessary to reorder the triangles to match them with a vertex and to pair them.

We also present a corresponding streaming reader that sequentially reads a Grouper stream and returns to the application a streaming mesh. Our reader preserves both vertex and triangle ordering. Both the reader and writer are compatible with the `libsm` streaming mesh API [31].

4.1 Grouper Construction: Streaming Writer

In this subsection, we describe a streaming process that constructs a Grouper representation from a streaming mesh consisting of an interleaved sequence of vertex and triangle records and vertex *finalization* tags. A finalization tag for vertex v arrives (at the earliest) with the last triangle incident upon v. After this happens, we say that v has been *finalized*. Note that an arriving triangle may finalize more than one vertex.

We first provide a brief overview of our approach. For convenience, we will use V* to refer to groups that contain a vertex (V, VT, and VTT), and T* to groups that contain at least one triangle (T, VT, and VTT). We make a distinction between "groups"—the conceptual constituents of a Grouper representation—and a "record"—the bytes representing a group. We maintain two data structures: a FIFO queue of entries for active groups and a buffer containing an active portion of the mesh. Each entry in the queue stores the group type (V, T, VT, or VTT) and either

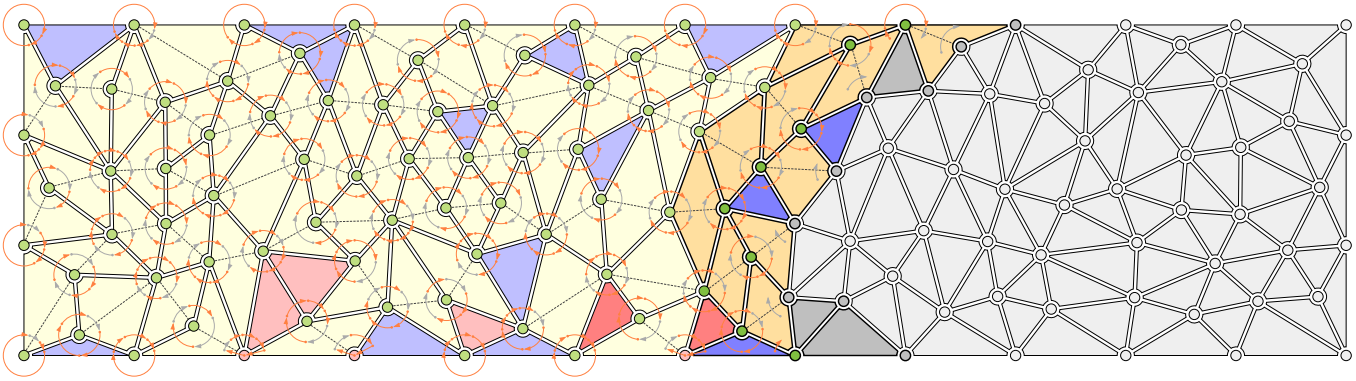


Fig. 5. Illustration of a partially constructed Grouper stream using our streaming writer. Light gray triangles and vertices have not yet been processed; saturated colors show active groups stored in the queue; while desaturated colors indicate groups that have been output. Triangles and vertices with thick outlines are active and are stored in the partial corner table. Stored (inter-group) swing pointers are shown as orange arrows; implicit (intra-group) swing pointers are gray. VTT groups are tan-colored triangles, VT are blue, T are red, and V are red vertices. This figure shows a single frame from an animation that is included with the article.

a single vertex index (for V) or canonical corner index (for T*), from which the remaining group corners can be inferred. Entries for V* groups are added at the end of the queue (initially as V groups) when the corresponding vertex arrives. Therefore, their order in the queue and hence in the output stream is the same as the order of the incoming vertices. Arriving triangles are added to the mesh buffer, which we examine to determine the adjacency of groups.

Each time a vertex v is finalized, we attempt to match v with one or two incident triangles present in the mesh buffer and also identify any incident **isolated triangles** that have no remaining vertices available for matching. For each isolated triangle we add an entry for a T group to the end of the queue. We say that a triangle is **mapped** when it is assigned to a group, either via matching or when placed in a T group, which establishes an index for the triangle in the output stream. For the entry at the front of the queue, we attempt to create an output record by testing if all of the group's swing targets have been mapped (i.e. have an assigned location in the output stream), in which case we say that the group is **complete**. For each complete group entry at the front of the queue we output a record.

The above process, illustrated in Fig. 5, is event driven, and computation is triggered each time a vertex, triangle, or finalization tag is encountered in the input stream. We detail each of these steps below.

Vertex Events To preserve the vertex order of the input mesh, when we encounter a new vertex we associate it with a new group and insert an entry for it in the queue. The input and output index (i.e. group index) of a vertex may nevertheless differ, because we interleave T records in the stream. Therefore, we store with each vertex of the mesh buffer its output index, i.e. the index of the corresponding record in the output stream.

We maintain a mesh buffer of active triangles and vertices until they are assembled into records and no longer needed. For simplicity we use a variation of Rossignac's Corner Table [32] that stores with each vertex v:

- a triplet of vertex coordinates,
- a reference $v.c$ to some incident corner,
- a bit to indicate whether v has been finalized,
- a reference count of incident triangles not yet output,
- an index identifying the record for v in the output stream.

Triangle Events When we encounter a new triangle in the input stream, we insert a corner for each of its vertices into our mesh buffer and for each such corner c we perform the following operations (1) For the vertex $c.v$ we increment a counter storing the number of incident corners. (2) We insert c as the head of a list of incident corners at the vertex $c.v$. This unordered list is stored as a temporary data structure using the swing references for each corner. The list starts at $v.c$, so we can insert in constant time. Later, when v is finalized, we fix the swing references to be consistent with their topological order around v. Thus, each triangle corner c is represented as:

- a reference $c.v$ to the corresponding vertex,
- a reference to the swing corner $c.s$,
- a bit to indicate if c is linked with $c.v$ (see below),
- an index for c in the output stream.

Finalization Events When a vertex v is finalized, we start a process of (1) assigning matched triangles to the V* group associated with v and, when necessary, creating T groups, and (2) streaming out complete group records whose swing references to adjacent groups have been determined. These two processes cannot be synchronized because a group record can only be streamed out when it is complete, i.e., when adjacent triangles have been incorporated into groups and proper inter-group swing references can be inferred. Hence, we keep a FIFO queue of incomplete groups.

We use a new matching process (different from the one proposed in [7]) to form groups from combinations of active vertices and triangles. As in SQuad, we strive to match each vertex with an adjacent pair of previously unmatched incident triangles. As explained in Section 6, this objective helps minimize total storage. Typically, some vertices end up being matched with a single triangle, and some vertices and triangles may remain unmatched. We experimented with several matching strategies, and concluded that eagerly matching a vertex with any adjacent pair of incident triangles, when possible, or otherwise with a single triangle worked as well as more sophisticated strategies, e.g. favoring matches with triangles with fewer unmatched vertices. Note that the constraints on matching boundary vertices imply that we must wait until v is finalized to perform matching, as only then can we determine whether v is a boundary vertex.

We link groups only around finalized vertices to ensure proper insertion of the V groups in the swing cycle. The V group of a boundary vertex v is linked between the last and the first corner around v, and hence acts as a virtual corner that completes the cycle around the vertex. The V group of an interior vertex v may in principle be linked between any pair of groups incident upon v. We say that the corner $v.c$ pointed to by a vertex v in a V group is **linked** with v.

The following sequence of steps is executed when a vertex v is finalized:

1. Update $c.s$ references around v. We use the initial $c.s$ references, which define an unsorted linked list of corners incident upon v, and rearrange these references into swing order around v. From now on, these $c.s$ references denote the proper swing of these corners.

2. Attempt to match v. We swing around v and search for a consecutive pair of unmatched incident triangles. When more than one pair is available, we use the first one found. If no pair exists, we match v with the first unmatched triangle in swing order, if one exists. If we cannot match v with an incident triangle, we leave it as a V group.

3. Identify isolated triangles. After matching, some triangles incident upon v may end up having all three of their vertices matched with other triangles. Such isolated triangles become T groups and we add entries for them at the end of the queue.

4. Output complete groups. We output records for complete group entries that are at the front of the queue and remove them from the queue. A V group is complete after its vertex has been finalized and linked with a mapped corner. T* groups are complete when their swing targets have been mapped.

5. Deallocate vertices and triangles. Once the records for a vertex and all its incident triangles have been output, no further references to the vertex are possible, and we deallocate the vertex to make room for new ones. Similarly, we deallocate a triangle when its vertices have been deallocated. We track this information using a reference counter stored with each vertex that is incremented each time an incident triangle arrives and decremented when the triangle is output in step 4.

4.1.1 Handling High-Span Streams

Because the Grouper construction algorithm preserves the input vertex ordering, the performance of operations on the resulting mesh depends upon having a streamable input ordering. In particular, the maximum size of the group buffer in both the reader and the writer is related to the largest swing distance, or equivalently to the maximum index difference between adjacent triangles. And because the triangle ordering is made "compatible" with the vertex ordering during matching, the size of the group buffer is a function of the *span* of the streaming mesh, i.e. the maximum index difference of active vertices [6]. This is so, because a group must be buffered until all of its swing pointers have been set, and we do not allow outputting groups out of order (for instance, groups further back in the queue that have been completed), in part because changing the location of a group would invalidate all swings to it from groups that have already been output.

A potential workaround would be to simply reserve a group and corner indices for the future adjacent triangle t swung to. However, this might fail, for instance because t's corner numbering depends on with which vertex it is matched, if any, and whether t is paired. Furthermore, this strategy assumes that t will appear in the near future, and indeed that it even exists, which may not be the case for meshes with boundaries.

We note that this buffering problem is not particular to our representation, but is true for any index-based graph structure with cycles. For instance, a streaming writer of Rossignac's Corner Table [32] or any other adjacency-based data structure suffers from the same problem.

In order to handle high-span streams without exhausting memory, we propose using **extensions**. An extension serves as a virtual copy of a record r that redirects any of r's unresolved swings, allowing a long (or simply unresolved) swing to be broken down into a sequence of shorter swings. Whenever record r at the front of the queue stays incomplete, e.g. because one of its incident vertices is not yet finalized, the queue continues to grow. When the queue size exceeds a user-specified limit, we evict r by first inserting an extension e at the end of the queue and then pointing any unresolved swing from r to the corresponding field in e. This allows r and any complete records waiting on r to be output. Once the extension reaches the front of the queue, we test if its deferred swings can now be resolved, and if not, create a second extension, thus further extending its swing loops.

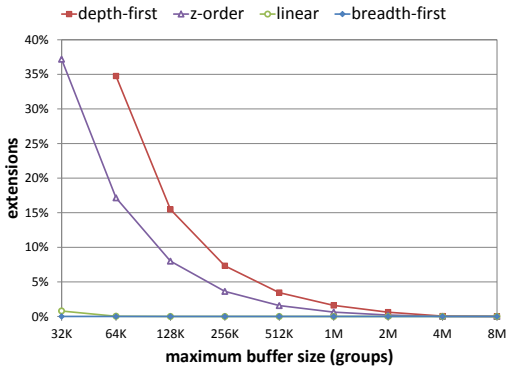


Fig. 6. The fraction of extension groups in the stream as a function of FIFO buffer size and layout of the 55 million triangle david mesh.

Although very long swings in high-span streams may have to be broken down into many extensions, such swings tend to be few when the width (maximum number of concurrently active vertices) of the stream is low; a precondition for frontal streaming [6]. Since each active vertex must be buffered, the buffer size must be at least as large as the width. In practice, using a queue of up to one million groups (our default) the number of extension groups tends to stay below 1–2% even for large high-span meshes (see Fig. 6). Because each FIFO entry is a single integer, even larger buffers can often be used to further limit the number of extensions.

Extensions are identified by our corner operators and iterators by having all of their corners marked as virtual, and never contain geometry. Virtual corners swung to in an extension are by convention assigned odd indices to maintain the efficiency of $c.v$, which as described in Section 3.1 traverses the loop looking for corner 0. Hence $c.v$ requires no modification to handle extensions. Similarly, $c.s$ already tests for and skips over virtual corners in V groups, and therefore needs no modification either.

4.2 Grouper Consumption: Streaming Reader

As a counterpart to the streaming writer described in Section 4.1, we describe here a corresponding streaming reader that reads a Grouper file or stream and emits to the application an interleaved sequence of vertices, indexed triangles (that refer to vertices by their global index), and finalization tags.

Our approach is to emit both vertices and triangles in the order in which they are stored in the Grouper stream by sequentially reading and buffering records in a FIFO queue. Whereas we may immediately pass through the vertex stored in a record to the application, the vertex indices $c.v$ associated with triangle corners c are not always readily available, as they are found by swinging to the corner matched with $c.v$, which may appear further along the stream. Consequently, before emitting the next triangle, we follow its swing pointers and attempt to complete loops. If in this process we reach a record that has not yet been read, then we sequentially input and buffer records and emit their vertices until the required record has been read. Once all records in the three swing loops of a triangle t are buffered, we say that t is **complete**, at which point we can infer its vertices and emit t.

The `libsm` API also provides one finalization bit per triangle corner that indicates whether the triangle is the last one in the stream to reference the associated vertex. When set, the application may deallocate storage for the vertex. Grouper does not explicitly store any finalization bits, because we can

infer this information by examining the swings around a vertex. In particular, when a closed swing loop of group records $L = (c.g, c.z.g, c.z.z.g, \ldots, c.g)$ is detected for $c.v$, all triangles incident on $c.v$ are known. (As discussed above, we also form loops for boundary vertices.) We thus mark $c.v$ as finalized by the last triangle stored in the (non-V) record in L with largest index. Although a triangle's vertices may in principle be known before their swing loops are closed, we do not emit the triangle until it is complete to ensure proper finalization. Based on this algorithm, we store with each record in the queue four entries—one per vertex—that each contain a swing pointer $c.z$, a vertex index $c.v$, and a finalization bit.

Once the vertex and triangles in the record at the front of the queue have been emitted, we can remove and deallocate the record. For a V group, we must first ensure that the swing loop it partakes in has been closed before removing it from the queue. Consequently, to locate the next triangle to be emitted, we maintain an additional pointer into the queue.

4.2.1 Handling High-Span Streams

Because the group record at the front of the buffer cannot be removed until its vertices have been finalized, the buffer may grow to be very large if the stream has a high span. This could be remedied simply by moving stagnant records to a spillover table, so that swings outside the buffer are redirected and looked up in the table.

5 Processing Grouper

Having described the basic Grouper representation, its construction, and operators acting upon it, we now turn our attention to how applications process Grouper streams.

5.1 Frontal Streaming

When memory is scarce or when the Grouper file does not reside on disk, e.g. arrives over a network or from another process, we advocate stream processing. In *frontal streaming*, the application maintains an active set of vertices—the *front*, consisting of introduced but not yet finalized vertices—which usually varies in size over the stream. Vertices are identified by global indices (e.g. when referenced by triangles), and it is customary to store the active vertices in a map (e.g. a hash). When a vertex is finalized, it can be removed from the map. Such a data structure is sufficient to, for instance, compute the surface area of a mesh.

For tasks that require processing not just individual triangles but larger collections of adjacent elements (e.g. simplification, subdivision, smoothing, etc.) the application also maintains an active set of triangles in a buffer between the input front and the output front (for applications that both read and write meshes). This usually involves dynamically inserting and removing triangles to and from a partial in-core mesh that supports full connectivity queries (cf. [6, 20, 28]). The triangle buffer may or may not be of fixed size. The dynamic memory management and construction of such an adjacency-based data structure can be quite computationally costly. In particular, this effort is duplicated by each module in a processing pipeline, multiplying the cost.

5.2 Windowed Streaming

Windowed streaming differs from frontal streaming in that a fixed-size buffer that holds a superset of the active vertices (and possibly active triangles) is used. Our implementation of windowed streaming maintains a circular fixed-size FIFO queue of records that acts as a sliding window over the mesh. Each incoming record replaces the least recently read record, which in a sense amounts to a conservative rather than eager approach to finalization. As long as the swing references are reasonably localized and never span more than the buffer size, this makes for a particularly efficient mode of processing. In particular, no mapping from the on-disk format to an in-core partial mesh data structure is needed (as in frontal streaming), because the two are one and the same. In case the buffer is too small and a swing reference points outside the buffer, a spillover buffer (e.g. a hash map) may be used, as suggested in [6]. Such "high-span" records are usually rare in otherwise streamable meshes with a low width, and therefore do not consume a lot of memory. Before evicting a record, one may determine if all of its vertices have been finalized by traversing their swing loops; a complete loop implies that a vertex can be finalized (even for boundary vertices; see Section 3.2). If not, the record is moved to the spillover buffer. We found that our benchmark meshes, when ordered along a single geometric direction, could be processed without a spillover buffer while using a sliding window smaller than 3% of the total mesh.

For tasks that only modify the geometry (e.g. smoothing), windowed streaming simply updates the geometry of each record and passes it through, possibly to a downstream module. In case the mesh connectivity is changed, we use the streaming writer from Section 4.1 to produce a new Grouper stream.

We note that hybrid frontal and windowed streaming approaches have been used previously, in which the sliding triangle buffer is of fixed size but the input and output vertex fronts are dynamically managed; see for instance [19, 20, 28]. Xia and Shaffer [29], on the other hand, make use of fixed-size vertex and tetrahedron buffers. Unlike Grouper, however, none of the mesh formats employed by these methods provide adjacency information, and therefore they all require on-the-fly connectivity reconstruction.

5.3 Out-of-Core Random Access

In addition to being a lean in-memory data structure, Grouper supports random-access traversals of meshes stored in external memory (i.e. disk) by memory mapping the mesh onto the operating system's virtual memory space, e.g. using the Linux `mmap` system call. This establishes a mapping between the on-disk mesh and the calling process's memory addresses, and enables *demand-driven paging* of the mesh from disk. This entails loading fixed-size, contiguous "pages" into memory whenever the application accesses a page that has either not yet been loaded or has been evicted from main memory.

On modern computers, the operating system augments demand-driven paging by predicting future data fetches and "prefetching" the associated pages. For mesh traversals that require random-access, prefetching is unlikely to predict future access, but for tasks like mesh smoothing that iterate over the whole mesh (e.g. using a sequential outer loop), accesses will proceed through the mesh in an approximately linear pattern, a use case for which prefetching systems are heavily optimized. Such sequential loops can trivially be parallelized on multicore computers using OpenMP [8] directives; something we explore in Section 7.

We note that although our $c.v$ operator may involve repeated memory accesses, it is often possible to confine such accesses to the same memory page using a coherent ordering of the vertices (and thus triangles). Moreover, the compactness of roughly 2 rpt for connectivity coupled with a compatible interleaving of geometry and connectivity promotes locality of reference in Grouper and avoids excessive thrashing.

Finally, one attractive property of Grouper is that the paging from external memory is done entirely transparently from the user application, which treats the Grouper data structure as though it were a complete contiguous in-memory array. Consequently, existing applications that rely on corner operators may use Grouper directly in in-core or out-of-core mode with no further code changes. In particular, such applications need not be re-engineered as stream modules, which might otherwise involve substantial algorithmic changes.

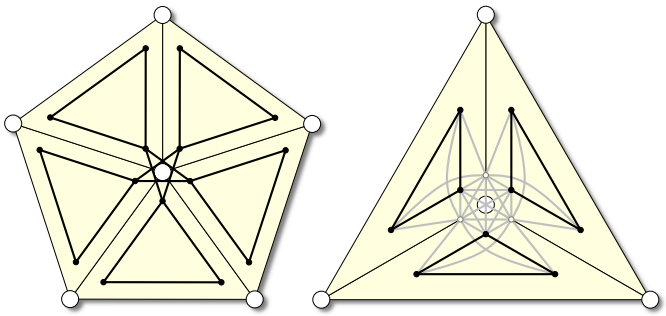


Fig. 7. Matching graph for general (left) and valence-three (right) vertices, which require additional nodes (hollow) and edges (gray).

6 Optimal Matching is NP-Hard

Our greedy matching and pairing algorithm strives to match one or two adjacent triangles with a shared vertex, and to leave as few unmatched triangles as possible. In an attempt to assess its effectiveness, we show that our problem is equivalent to the well-known *maximum independent set* (MIS) problem, and compare our solution to prior heuristics for MIS. Given an undirected graph, the MIS problem is to find a set of mutually non-adjacent nodes of maximum cardinality; a problem known to be NP-hard.

The matching and pairing needed for Grouper can be cast as a combinatorial optimization problem in which we seek to minimize the storage cost. It is easy to see that the storage cost in number of fixed-size records (excluding extensions) equals $n_V + n_U$, where n_V is the number of vertices and n_U the number of unmatched triangles (i.e. T groups). This is so, because each record either stores a vertex or an unmatched triangle. Since n_V is fixed, our goal is to minimize the number of unmatched triangles n_U.

We reduce the matching problem to MIS by constructing a graph $G = (C, E)$ from a triangle mesh. The nodes C correspond to the corners of the mesh. Thus each corner added to the independent set corresponds to matching a vertex with a triangle. To avoid multiple matches per triangle, we include in E an edge between each pair of corners within a triangle. To prevent matching vertices with more than two edge-adjacent triangles, we add edges between each pair of non-adjacent corners (c_i, c_j) around the vertex, i.e. corners such that $c_i.v = c_j.v, c_i.s \neq c_j, c_j.s \neq c_i$. Because no edges exist between adjacent corners, pairs of edge-adjacent triangles may still be matched with a shared vertex.

The resulting graph is illustrated in Fig. 7(left). It has all the right properties except for interior vertices of valence three. Because the corners $\{c_i, c_j, c_k\}$ around such vertices are mutually adjacent, nothing prevents all three of them from being matched with the same vertex. We resolve this by adding to C three additional nodes $\{c_{ij}, c_{jk}, c_{ki}\}$ that represent pairs of triangles, letting the existing corners denote unpaired triangles. We also add edges between all nodes $\{c_i, c_j, c_k, c_{ij}, c_{jk}, c_{ji}\}$ and between c_{ij} and the existing neighbors of c_i and c_j (and similarly for c_{jk} and c_{ki}), for a total of 27 additional edges. The maximum independent set of this graph then represents the matching and pairing with the lowest storage cost.

We note that since there are roughly twice as many triangles as vertices, it is in general not possible to minimize the number of unmatched triangles without also pairing triangles. In other words, the MIS solution will favor matching shared over non-shared vertices in adjacent triangles, since doing so "costs" only one instead of two precious matched vertices.

We have compared Grouper with several heuristic algorithms for MIS [33–36]. Our algorithm is able to exploit the particular structure of the graph, and therefore leaves far fewer triangles unmatched than the more naïve MIS algorithms Basic and RandomOffLine described in [36] and the well-known minimum-degree Greedy algorithm (see [34]). On average, we produce fewer than 3% unmatched triangles. For small meshes with a few hundred triangles, however, we have found that GRASP [34] and Dharwadker's algorithm [35] may produce optimal solutions with no unmatched triangles. Given their high asymptotic complexity (e.g. the latter runs in $O(n^8)$ time on n nodes), their use on larger meshes may be limited.

7 Results

We here present storage and performance results for our Grouper representation. We used an 8-core 2.66 GHz Intel Xeon X5550 computer with 12 GB of 1.33 GHz DDR3 RAM and a 7,200 RPM Seagate Barracuda SATA disk for our experiments.

7.1 Storage Efficiency

In order to assess how well we are able to match and pair triangles, we used our streaming writer with a FIFO queue of at most one million groups to convert several meshes ordered in a number of ways, including breadth- and depth-first orderings, and geometric orderings based on linearly sorting along a single direction and by ordering vertices along a z-order space-filling curve. We use 32-bit floats and integer references and assume a fixed geometry storage cost of 3 floats per vertex, or equivalently 1.5 rpt (references per triangle) using the approximation $n_T \approx 2n_V$. The connectivity storage cost of our Grouper representation is $4(n_V + n_U)$ rpt, or roughly 2.0 rpt when n_U (the number of unmatched triangles) is small. In addition to storage efficiency, we present in Table 1 the overhead component n_U/n_V of the storage cost in records stored per vertex (recall that we must store at least n_V records). To calibrate our results, we compare storage (including geometry) with the SMB binary streaming mesh format [31]; a variation on Rossignac's Corner Table [32] dubbed VST (for vertex swing table); and meshes constructed by the original in-memory SQuad method [7], which has more flexibility when matching because it can choose the traversal order. We also compare our overhead with the minimum-degree greedy independent set algorithm [33]. Our comparisons include only representations that use fixed-length encodings, and a more comprehensive comparison against representations with variable-length encodings, e.g. OEMM [4], OoCM [20], SMC [21], or RACM [13] is left as future work.

The SMB format stores vertex coordinates and indices as 32-bit words. To distinguish vertices from triangles, one bit per mesh element is used. These bits are packed into 32-bit words that specify the type of the next 32 records. One bit per vertex index is reserved for finalization. The resulting storage cost (geometry + connectivity) is roughly $1.5 + 3.0 = 4.5$ rpt. As seen in the table, our format reduces storage over SMB by about 20%, while encoding incidence and finalization, and (additionally) adjacency.

The VST format stores with each triangle corner $c.v$ and $c.s$, and with each vertex $v.c$, resulting in a total cost of roughly $1.5+6.5 = 8.0$ rpt. Consequently VST requires about 2.2 times the storage of Grouper, while not being readily streamable.

Because SQuad favors pairing adjacent consecutive triangles in a spiraling traversal, its storage cost is similar to our Grouper when given a mesh in a similar depth-first order. One notable exception is the Puget Sound mesh, whose relatively high width and span resulted in 15% of records being extensions. With no FIFO queue limit, its storage cost is reduced to 2.172 rpt. The geometrically ordered meshes arrive in a less predictable (to the writer) order, and with less locality of reference. In particular, the z-ordered meshes are rich in high-span vertices that have an adverse effect on matching. Nevertheless, even the worst re-

mesh	n_T	%v6	breadth-first		depth-first		linear		z-order		SQuad	Greedy	SMB	VST
			rpt	ovhd	rpt	ovhd	rpt	ovhd	rpt	ovhd	rpt	ovhd	ratio	ratio
bunny	69,451	75.1	2.060	2.7	2.061	2.7	2.066	3.0	2.132	6.2	2.054	28.5	1.26	2.22
rocker arm	80,354	65.2	2.059	2.9	2.055	2.7	2.051	2.5	2.115	5.7	2.052	28.3	1.26	2.22
horse	96,966	66.5	2.058	2.9	2.055	2.7	2.061	3.1	2.120	6.0	2.046	28.2	1.26	2.22
dinosaur	112,384	57.9	2.081	4.0	2.078	3.9	2.084	4.2	2.161	8.1	2.072	27.6	1.25	2.20
armadillo	345,944	52.6	2.073	3.6	2.075	3.7	2.096	4.8	2.139	6.9	2.069	27.0	1.25	2.21
hand	654,666	53.4	2.092	4.6	2.091	4.5	2.096	4.8	2.163	8.1	2.096	27.0	1.24	2.19
buddha	1,087,716	32.1	2.186	9.4	2.178	9.0	2.171	8.6	2.221	11.1	2.150	25.3	1.19	2.09
blade	1,765,388	62.4	2.081	4.0	2.086	4.3	2.088	4.4	2.123	6.1	2.077	27.7	1.25	2.20
welsh dragon	2,210,673	86.7	2.026	1.3	2.033	1.7	2.031	1.5	2.111	5.5	2.027	29.7	1.28	2.26
asian dragon	7,219,045	89.1	2.033	1.7	2.039	1.7	2.026	1.3	2.115	5.7	2.026	29.8	1.28	2.25
thai statue	10,000,000	44.4	2.136	6.8	2.220	7.2	2.123	6.2	2.222	11.0	2.111	26.3	1.22	2.14
david	55,514,795	51.6	2.056	2.6	2.096	2.9	2.062	2.9	2.163	7.3	2.082	26.7	1.26	2.22
puget sound	134,217,728	29.3	2.154	7.7	2.580	9.5	2.151	7.5	2.193	8.3	2.156	24.5	1.21	2.12
median	1,087,716	57.9	2.073	3.6	2.078	3.7	2.084	4.2	2.139	6.9	2.072	27.6	1.25	2.21
mean	16,413,470	58.9	2.084	4.2	2.127	4.3	2.085	4.2	2.152	7.4	2.078	27.4	1.25	2.20

Table 1. We report for each mesh its number of triangles n_T and percentage of valence-6 vertices. For each layout, we report Grouper connectivity storage in references per triangle (rpt) and percentage overhead n_U/n_V in terms of unmatched triangles n_U. We also report corresponding results for SQuad, greedy independent sets, and the total storage ratio of SMB and VST to Grouper for the breadth-first layout.

sults of our method show a significant improvement in matched triangles over the greedy MIS method.

7.2 I/O Speed

We construct a Grouper representation from streaming input at a rate of about 1.7 million triangles per second (excluding I/O time). For low-span streams (e.g. breadth-first and linear geometric orderings) the construction buffers in memory a maximum of 1–3% of the input triangles. For example, the 134 million triangle Puget Sound mesh can be converted from a 2.3 GB SMB file to a 1.9 GB file in our Grouper format using only 9.8 MB of working memory. This compares favorably with the SQuad in-core construction, which for the same mesh uses 6.2 GB of RAM and takes twice as long.

Our frontal streaming reader of Grouper files described in Section 4.2 achieves a throughput of about 2.8 million triangles per second (including I/O time), which is roughly two times slower than reading an SMB file. This difference in speed is attributable to the need to buffer records, to resolve triangle-to-vertex references, and to detect finalization events using our representation. As shown below, I/O and processing of Grouper are more efficiently accomplished using windowed rather than frontal streaming.

7.3 Operator Speed

At 3.9 nanoseconds, our $c.s$ operator is comparable in speed to its SQuad counterpart. In spite of $c.v$ being conceptually simpler, S array accesses in Grouper involve converting relative indices to absolute ones that are then implicitly remapped to skip over the interleaved geometry records, resulting in a 14.1 nanosecond execution time, or 45% slower than in SQuad.

7.4 Processing Speed

We evaluated several different modes of accessing our Grouper format using a suite of processing kernels designed to exercise a variety of mesh queries and traversal patterns:

- **Components**: Count the number of connected components in the primal graph. This is accomplished using a generalization of the algorithm presented in [37], which uses incidence information only and a union-find forest that is pruned each time a vertex is finalized.

- **Area**: Loop over all triangles and compute the total surface area. This can be computed directly by maintaining a map of vertex coordinates keyed by global vertex index, i.e. without maintaining any connectivity.

- **Silhouette**: Count the number of silhouette edges with respect to an arbitrary view direction. For each incident edge of each finalized vertex v, we compute the normals of the two adjacent faces that share the edge. We then compare the signs of the dot products of the normals with the view direction to determine if the edge is on the silhouette. Note that we count these silhouettes even if they are occluded.

- **Traversal**: Starting from an arbitrary seed triangle, perform a spiraling depth-first traversal of the whole mesh by visiting adjacent triangles (as in [32]). To speed up the computation and keep the stack size small, we maintain an auxiliary visitation flag with each vertex and triangle.

- **Ascent**: Starting from an arbitrary seed vertex, perform a steepest ascent traversal along the mesh edges using one of the coordinates as function value. The traversal ends at a local maximum.

- **Geodesic**: Starting from an arbitrary seed point and direction, trace a geodesic path along the surface until a surface boundary is encountered.

For the first three tasks, a single-pass streaming implementation is possible, which we describe briefly above. A corresponding random-access implementation loops over vertices or triangles in index order, and for connected components infers finalization information by testing if the current triangle has the highest index among those triangles incident on a vertex. (Because such finalization information allows the union-find data structure to be pruned, this random-access approach to computing connected components is faster than alternative methods, e.g. based on invading and marking the vertices of each component.) The remaining three tasks traverse the mesh in a data-dependent manner dictated by the mesh geometry and/or connectivity, precluding a single-pass streaming implementation and necessitating random access. The last two tasks, in particular, visit only a small subset of the mesh. In these cases, we executed the task many times using different seeds in order to obtain reliable timings.

In addition to the mesh queries involved in these tasks, Table 2 summarizes the median execution time across 15 runs for each task, data structure, and access pattern. We note that the execution time depends largely on whether the mesh has to be fetched from disk or is already partially or even totally memory-resident. The latter case occurs, for instance, in pipelined streaming, when the output of one process is piped directly to the input of the next downstream process via shared memory.

task	geo.	adj.	sub.	cold mode							warm mode						
				VST	direct	interleaved	blocked	SMB	frontal	windowed	VST	direct	interleaved	blocked	SMB	frontal	windowed
components				47.5	36.4			19.3	26.4	26.4	25.6	23.3			12.5	20.5	21.4
area	✓			52.9	24.7	15.9	27.7	15.3	23.6	15.9	11.5	11.3	7.0	3.2	8.1	17.9	9.7
silhouette	✓	✓		64.3	35.7	14.9	40.1	42.5	46.9	25.8	23.2	22.5	7.0	3.5	36.8	41.6	20.9
traversal		✓		96.3	48.3						14.7	12.6					
ascent	✓	✓	✓	72.5	30.4						17.3	11.8					
geodesic	✓	✓	✓	8.6	6.1						5.0	4.8					

Table 2. Median execution time in seconds for our benchmark tasks using different mesh representations and access patterns. Only the top three tasks have streaming implementations. The leftmost columns show the mesh queries (geometry and adjacency) involved and whether only a subset of the mesh is visited. We report timings for cold mode (disk caches are initially flushed) and warm mode (disk caches are pre-loaded). Columns other than **VST** and **SMB** correspond to the use of Grouper in **direct** (random-access) serial mode and in parallel mode with **interleaved** and contiguous **block** static loop scheduling, as well as using **frontal** and **windowed** streaming.

The mesh may also be partially cached in disk buffers from earlier processing. Consequently, we timed each task both when the disk buffers were explicitly flushed before each run (cold mode) using the Linux `drop_caches` mechanism, and when the caches were warmed by first executing the task once and then timing the next 15 runs without flushing (warm mode). These two modes can be thought of as extremes that provide lower and upper bounds on processing time. Moreover, by running in warm mode, we are able to largely exclude the dominant disk access time, allowing us to measure the underlying performance of each data structure independent of any speedups obtained through reduced I/O.

In order to support direct random access to the Grouper file, we used memory mapping (both in cold and warm mode). The area and silhouette computations loop over the mesh triangles or vertices by index, and trivially parallelize. We used OpenMP for loop parallelization with static scheduling and two different assignments of threads to loop iterations: *interleaved*, in which thread i of n processes indices $kn + i$ (where k is a non-negative integer), and *blocked*, in which each thread is assigned an equal-size contiguous subsequence of indices.

To evaluate the different mesh representations, we used the David mesh in breadth-first order. We converted this mesh from SMB format to our Grouper format, which very slightly changed the order of triangles to accommodate matching. The resulting Grouper representation was then converted to VST, preserving the ordering of both vertices and triangles (but removing "holes" in indices due to non-VTT records).

Based on the numerical results from Table 2, also shown graphically in Fig. 8, we make the following observations.

In direct access mode on a single processor (Fig. 8(d)), Grouper yields improved performance over VST—even when the mesh is memory-resident. In cold mode, Grouper is 1.3–2.4 times faster. Although all tasks visit the same number of triangles, notice that the geodesic task is much faster than the others because it often revisits the same cached subset of triangles, e.g. when circling a cylindrical part of the mesh, such as an arm.

Using OpenMP for loop parallelization on 8 cores, we achieved speedups of up to 6.4x. As is evident from Fig. 8(b) and 8(c), interleaved thread assignment is beneficial in cold mode, as this gives each thread some amount of work to do each time a disk block is loaded, while in blocked mode the threads contend for I/O and are idle while waiting for I/O requests to be serviced. In warm mode, the roles are reversed, in part because of the drastic differences in latency and bandwidth between memory and disk accesses. Furthermore, loop blocking provides for higher locality of reference and thread-local caching, whereas when interleaved the threads each touch every single memory page of the mesh via small strides, and in effect increase memory bandwidth and cache usage by the number of threads.

In cold frontal streaming mode, Grouper is 1.1–1.5 times slower than SMB. Because the frontal streaming implementations are built on top of the same streaming mesh API, the same tasks are executed for both formats, possibly including connectivity reconstruction, which does not take advantage of the adjacency information stored with Grouper. As discussed above, our Grouper frontal reader must also buffer records and recover vertex references and finalization—information that is readily available in the SMB format. This extra work can be costlier than the simple processing tasks themselves.

We further note that frontal streaming access is more efficient than random access using Grouper for tasks that require only incidence (i.e. components and area). However, when adjacency information is needed, as in the silhouette computation, frontal streaming is less efficient, because then the adjacency information must first be recovered via construction and dynamic management of a partial mesh data structure. Note that Grouper already provides this information in direct mode.

Using Grouper, windowed streaming outperforms frontal streaming by as much as a factor of two. In spite of having to explicitly maintain a circular buffer of records (we used a fixed-size buffer of 2^{16} records) and incurring one disk read per record, windowed streaming is also consistently more efficient than memory-mapped direct access, both in cold and warm mode, in part due to a much smaller memory footprint. This performance difference is particularly evident in cold mode, where the sequential reads made in windowed streaming allow the operating system to pre-fetch disk blocks. Although the outer loop in direct mode is also sequential, the non-sequential accesses made to incident elements, e.g. when resolving vertex references by following swing loops, result in localized but non-sequential accesses, making it more difficult for the operating system to predict the next memory page needed.

8 Discussion

Before concluding this paper, we summarize some of the limitations and benefits of our Grouper representation.

8.1 Limitations

We envision Grouper being most useful in algorithms that do not require many "random" changes to connectivity. Though it is possible to reconstruct groups within the local neighborhood of a changed triangle, if the number of groups increases the new records must be placed at the end of the array, negatively impacting locality of reference and overall performance. For processing that results in a completely new mesh, such as subdivision or decimation, our streaming writer may be used to output the mesh. High-span streams like depth-first and space-filling orderings require the use of extensions, which incur an overhead in storage.

One possible drawback of our representation is that both vertex and triangle indices contain "holes" that correspond to non-VTT groups. Although such holes are easy to identify, applications that assume contiguous indices must be modified. Moreover, while rare (less than 5% on average), these holes lead to an overhead in storage for any user-defined data associated with vertices or triangles.

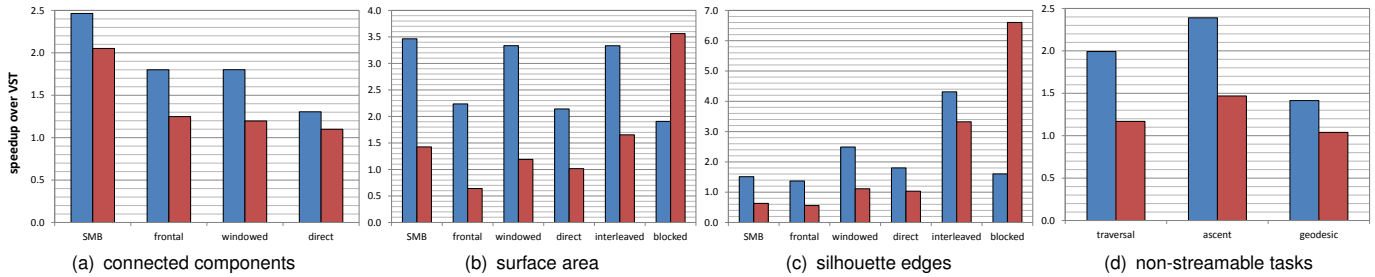


Fig. 8. Performance of (a–c) streamable tasks and (d) Grouper direct access relative to the VST random-access mesh data structure in cold (blue) and warm (red) mode.

8.2 Benefits

Our Grouper representation offers the following benefits over alternative data structures and file formats:

- Like SMB, Grouper is a streaming mesh representation. However, Grouper also supports random access to vertices and triangles, and directly stores adjacency information that an SMB reader must recover on the fly. In spite of this, Grouper uses roughly 20% less storage than SMB.
- Although VST and Grouper both support random access, VST does not interleave geometry and connectivity, and therefore does not support (linear) stream processing. This interleaving coupled with relative indexing further enables instant partitioning for distributed data-parallel processing of Grouper without reindexing or repackaging. Moreover, Grouper uses 2.2 times less storage than VST, resulting in a proportional performance increase in out-of-core applications.
- Unlike SQuad, Grouper supports a memory-efficient construction process that allows the mesh to be streamed out immediately during mesh generation or editing. Grouper also enforces the locality of reference that SQuad lacks and is needed for subsequent stream processing. Finally, Grouper stores triangle rather than quad corner references, which results in simpler mesh navigation operators.
- In comparison to out-of-core representations like OEMM [4] and Isenburg and Gumhold's out-of-core mesh (OoCM) [20], Grouper supports streaming construction and more efficient storage, thereby reducing time, temporary and persistent disk usage, and memory usage during construction. For instance, Grouper uses 2.4 times less storage than both OEMM and OoCM for the David mesh, and is constructed roughly 100 and 42 times faster than indexed OEMM and OoCM, respectively.

In addition to serving as the first unified mesh file format and data structure for both streaming and random access, we have found Grouper useful as a compact intermediate representation for constructing even more space-efficient mesh data structures, such as the recently proposed LR [9] and Zipper [10] data structures. Building such data structures requires a temporary mesh representation that supports adjacency and incidence queries, but usually the input stores only incidence information. Because Grouper requires less storage than a standard incidence-based file format like SMB, and because it can be converted from such a format using very little memory, it is suitable as both input and temporary representation for constructing and possibly even rebuilding mesh data structures in applications that generate or modify the mesh connectivity.

Finally, a unique strength of Grouper is the support for both streaming and random access through already established APIs. This allows Grouper to be used in existing streaming or random-access applications with no further code changes.

9 Conclusion

We have presented Grouper: a data structure and format for representing triangle meshes that provides adjacency and incidence information in amortized constant time and that interacts well with virtual memory and processor caches. Our format supports the `libsm` streaming mesh API, making it a drop-in replacement for existing streaming algorithms. It also supports random-access mesh traversal, making possible out-of-core algorithms that are difficult to write in a streaming paradigm. Grouper enables parallel processing by allowing multiple threads to iterate over a mesh without the synchronization bottleneck created when allocating and deallocating vertex storage in a typical streaming algorithm. We have presented a construction algorithm for our data structure that operates on streaming input and that produces meshes whose connectivity storage, at just over two references per triangle, rival those attained by the global, non-streaming SQuad construction algorithm. We identify the construction problem as a well-known NP-hard optimization problem, and show that our algorithm is an excellent heuristic solution. By relaxing the order in which vertices and triangles are stored, we are able to improve locality of reference over SQuad, thereby enabling memory-efficient streaming and more general out-of-core computations on huge meshes using SQuad's compact storage.

Acknowledgements

This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[1] Shyh-Kuang Ueng and Krzysztof Sikorski, "An out-of-core method for computing connectivities of large unstructured meshes", in *Eurographics Workshop on Parallel Graphics and Visualization*, 2002, pp. 97–103.

[2] Christopher DeCoro and Renato Pajarola, "XFastMesh: Fast view-dependent meshing from external memory", in *IEEE Visualization*, 2002, pp. 363–370.

[3] Martin Isenburg and Stefan Gumhold, "Out-of-core compression for gigantic polygon meshes", *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 935–942, 2003.

[4] Paolo Cignoni, Claudio Montani, Claudio Rocchini, and Roberto Scopigno, "External memory management and simplification of huge meshes", *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 4, pp. 525–537, 2003.

[5] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno, "Adaptive tetrapuzzles: Efficient out-of-core construction and visualization of gigantic multiresolution polygonal models", *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 796–803, 2004.

[6] Martin Isenburg and Peter Lindstrom, "Streaming meshes", in *IEEE Visualization*, 2005, pp. 231–238.

[7] Topraj Gurung, Daniel Laney, Peter Lindstrom, and Jarek Rossignac, "SQuad: Compact representation for triangle meshes", *Computer Graphics Forum*, vol. 30, no. 2, pp. 355–364, 2011.

[8] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon, *Parallel programming in OpenMP*, Academic Press, 2001.

[9] Topraj Gurung, Mark Luffel, Peter Lindstrom, and Jarek Rossignac, "LR: Compact connectivity representation for triangle meshes", *ACM Transactions on Graphics*, vol. 30, no. 4, pp. 67:1–67:8, 2011.

[10] Topraj Gurung, Mark Luffel, Peter Lindstrom, and Jarek Rossignac, "Zipper: A compact connectivity data structure for triangle meshes", *Computer Aided Design*, vol. 45, no. 2, pp. 262–269, 2013.

[11] Jack Snoeyink and Bettina Speckmann, "Tripod: A minimalist data structure for embedded triangulations", in *Computational Graph Theory and Combinatorics*, 1999.

[12] Sung-Eui Yoon, Brian Salomon, Russell Gayle, and Dinesh Manocha, "Quick-VDR: Interactive view-dependent rendering of massive models", in *IEEE Visualization*, 2004, pp. 131–138.

[13] Sung-Eui Yoon and Peter Lindstrom, "Random-accessible compressed triangle meshes", *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1536–1543, 2007.

[14] Clement Courbet and Celine Hudelot, "Random accessible hierarchical mesh compression for interactive visualization", *Computer Graphics Forum*, vol. 28, no. 5, pp. 1311–1318, 2009.

[15] Sungyul Choe, Junho Kim, Haeyoung Lee, and Seungyong Lee, "Random accessible mesh compression using mesh chartification", *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 1, pp. 160–173, 2009.

[16] Yi-Jen Chiang and Cláudio T. Silva, "I/O optimal isosurface extraction", in *IEEE Visualization*, 1997, pp. 293–300.

[17] Peter Lindstrom, "Out-of-core simplification of large polygonal models", in *ACM SIGGRAPH*, 2000, pp. 259–262.

[18] Sara McMains, Joseph M. Hellerstein, and Carlo H. Séquin, "Out-of-core build of a topological data structure from polygon soup", in *ACM Symposium on Solid modeling and Applications*, 2001, pp. 171–182.

[19] Jianhua Wu and Leif Kobbelt, "A stream algorithm for the decimation of massive meshes", in *Graphics Interface*, 2003, pp. 185–192.

[20] Martin Isenburg, Peter Lindstrom, Stefan Gumhold, and Jack Snoeyink, "Large mesh simplification using processing sequences", in *IEEE Visualization*, 2003, pp. 465–472.

[21] Martin Isenburg, Peter Lindstrom, and Jack Snoeyink, "Streaming compression of triangle meshes", in *Eurographics Symposium on Geometry Processing*, 2005, pp. 111–118.

[22] Martin Isenburg, Yuanxin Liu, Jonathan Shewchuk, and Jack Snoeyink, "Streaming computation of Delaunay triangulations", *ACM Transactions on Graphics*, vol. 25, no. 3, pp. 1049–1056, 2006.

[23] Minsu Ahn, Igor Guskov, and Seungyong Lee, "Out-of-core remeshing of large polygonal meshes", *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1221–1228, 2006.

[24] Valerio Pascucci, Giorgio Scorzelli, Peer-Timo Bremer, and Ajith Mascarenhas, "Robust on-line computation of Reeb graphs: Simplicity and speed", *ACM Transactions on Graphics*, vol. 26, no. 3, pp. 58, 2007.

[25] Huy T. Vo, Steven P. Callahan, Peter Lindstrom, Valerio Pascucci, and Cláudio T. Silva, "Streaming simplification of tetrahedral meshes", *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 1, pp. 145–155, 2007.

[26] Rémi Allègre, Raphaëlle Chaine, and Samir Akkouche, "A streaming algorithm for surface reconstruction", in *Eurographics Symposium on Geometry Processing*, 2007, pp. 79–88.

[27] Matthew Bolitho, Michael Kazhdan, Randal Burns, and Hugues Hoppe, "Multilevel streaming for out-of-core surface reconstruction", in *Eurographics Symposium on Geometry Processing*, 2007, pp. 69–78.

[28] Pablo Diaz-Gutierrez, Jonas Bösch, Renato Pajarola, and M. Gopi, "Streaming surface sampling using Gaussian ϵ-nets", *The Visual Computer*, vol. 25, no. 5–7, pp. 411–421, 2009.

[29] Tian Xia and Eric Shaffer, "Streaming tetrahedral mesh optimization", in *ACM Symposium on Solid and Physical Modeling*, 2008, pp. 281–286.

[30] M. Mantyla, *Introduction to Solid Modeling*, W. H. Freeman & Co., 1988.

[31] Martin Isenburg and Peter Lindstrom, "libsm", 2005, `http://www.cs.unc.edu/~isenburg/sm/`.

[32] Jarek Rossignac, Alla Safonova, and Andrzej Szymczak, "Edgebreaker on a corner table: A simple technique for representing and compressing triangulated surfaces", in *Hierarchical and Geometrical Methods in Scientific Visualization*, pp. 41–50. Springer Verlag, 2003.

[33] Magnús Halldórsson and Jaikumar Radhakrishnan, "Greed is good: Approximating independent sets in sparse and bounded-degree graphs", in *ACM Symposium on Theory of Computing*, 1994, pp. 439–448.

[34] Mauricio G. C. Resende, Thomas A. Feo, and Stuart H. Smith, "Algorithm 787: Fortran subroutines for approximate solution of maximum independent set problems using GRASP", *ACM Transactions on Mathematical Software*, vol. 24, no. 4, pp. 386–394, 1998.

[35] Ashay Dharwadker, "The independent set algorithm", 2006, `http://www.dharwadker.org/independent_set/`.

[36] Bjarni V. Halldórsson, Magnús M. Halldórsson, Elena Losievskaja, and Mario Szegedy, "Streaming algorithms for independent sets", in *International Colloquium on Automata, Languages and Programming*, 2010, pp. 641–652.

[37] Martin Isenburg and Jonathan Schewchuk, "Streaming connected component computation for trillion voxel images", in *Workshop on Massive Data Algorithmics*, 2009.